Tracking Normalized Network Traffic Entropy to Detect DDoS Attacks in P4

Damu Ding, *Student Member, IEEE*, Marco Savi, and Domenico Siracusa

Abstract—Distributed Denial-of-Service (DDoS) attacks represent a persistent threat to modern telecommunications networks: detecting and counteracting them is still a crucial unresolved challenge for network operators. DDoS attack detection is usually carried out in one or more central nodes that collect significant amounts of monitoring data from networking devices, potentially creating issues related to network overload or delay in detection. The dawn of programmable data planes in Software-Defined Networks can help mitigate this issue, opening the door to the detection of DDoS attacks directly in the data plane of the switches. However, the most widely-adopted data plane programming language, namely P4, lacks supporting many arithmetic operations, therefore, some of the advanced network monitoring functionalities needed for DDoS detection cannot be straightforwardly implemented in P4. This work overcomes such a limitation and presents two novel strategies for flow cardinality and for normalized network traffic entropy estimation that only use P4-supported operations and guarantee a low relative error. Additionally, based on these contributions, we propose a DDoS detection strategy relying on variations of the normalized network traffic entropy. Results show that it has comparable or higher detection accuracy than state-of-the-art solutions, yet being simpler and entirely executed in the data plane.

Index Terms—Network monitoring, Programmable data planes, P4, Normalized network traffic entropy, DDoS detection

1 Introduction

Distributed Denial-of-Service (DDoS) attacks are becoming one of the most significant threats for network operators and their customers as such attacks, carried out by many different compromised hosts, are able to flood a victim with a huge load of superfluous traffic and exhaust its network and computational resources, causing service disruptions. In this context, detecting DDoS attacks in a smooth yet effective way plays a key role in today's network security. Periodical monitoring of specific network metrics has been widely adopted as a strategy to detect DDoS attacks. For instance, network traffic entropy is a statistical measure to describe the flow distribution, and the entropy of distinct destination IPs observed in the network significantly decreases during a DDoS attack [2][3][4]. Moreover, a significant increase in the number of source IPs contacting a specific destination IP [5][6][7] may also indicate that a DDoS attack is taking place.

From a technological perspective, both in SNMP-based [8] networks and in more recent Openflow-based [9] Software-Defined Networks (SDNs), monitoring data collection and, consequently, DDoS detection are carried out by a logically centralized component (generally known as *monitoring collector* or, more widely, *controller*): this requires the transmission, storage, and processing of a large amount of information related to the network state from network devices to this component [10]. Such an approach comes with two well-known drawbacks [11]: *(i)* a significant communication overhead is generated between data and centralized monitoring/control planes and *(ii)* significant processing capabilities are needed by the collector, with the risk of affecting the performance of monitoring and network operations if involved parties are not well-dimensioned.

The recent advent of so-called (data-plane) *programmable switches* allows network operators to partially overcome such drawbacks. In fact, programmable switches can, if appropriately programmed, execute part of the network monitoring/security operations directly in their data plane pipeline and deliver to the centralized monitoring/control plane information that is partially or fully processed. However, data-plane programming comes with some inherent limitations: the most well-established and widely-adopted data-plane programming language, called P4 [12], does not natively support basic yet relevant arithmetic operations such as division, logarithm and exponential function calculation, as well as any operation on floating numbers or *for* loops. Unfortunately, all these operations are needed to effectively implement an *entropy-based DDoS detection* strategy that *(i)* is able to evaluate abnormal variations on the entropy over time and *(ii)* can be fully executed in the programmable data plane, that is, it operates within the data plane pipeline and forwards alarms to the monitoring collector to notify about potential DDoS attacks.

However, spotting variations of entropy over time, as done in previous works, may not be the most effective way to detect DDoS attacks. In fact, the number of distinct flows in the network (i.e., *flow cardinality*) changes dynamically, affecting, in turn, the value of traffic entropy. A more suitable metric is therefore the *normalized entropy*, which is normalized against flow cardinality and is more robust to legitimate changes on the number of distinct flows.

The goal of this paper is thus to propose novel strategies

Damu Ding was with Fondazione Bruno Kessler, Trento, Italy and University of Bologna, Bologna, Italy. He is now with University of Oxford. E-mail: damu.ding@eng.ox.ac.uk. Marco Savi is with University of Milano-Bicocca, Milano, Italy. E-mail: marco.savi@unimib.it. Domenico Siracusa is with Fondazione Bruno Kessler, Trento, Italy. E-mail: dsiracusa@fbk.eu. *A preliminary version of this paper appeared in [1], presented at IEEE/IFIP NOMS in 2020. The research leading to these results has received funding from the EC within the H2020 Research and Innovation program, Grant Agreement No. 856726 (GN4-3 project).*

to estimate network traffic statistics such as normalized entropy and flow cardinality directly in P4 programmable switches, with the final goal of using them as building blocks to accurately and timely detect DDoS attacks. To this aim, based on P4-based solutions for the estimation of logarithm (P4Log) and exponential function (P4Exp) that we proposed in a preliminary version of this work [1], we here propose *P4LogLog*, a novel memory-efficient strategy that takes inspiration from LogLog algorithm [13] for the estimation of flow cardinality in P4. We then present *P4NEntropy*, our strategy for normalized Shannon entropy [14] estimation in P4, and *P4DDoS*, our approach for DDoS detection based on P4NEntropy. Even though we designed and implemented P4LogLog and P4NEntropy in support to P4DDoS, they can be seen as two stand-alone strategies paving the way towards the development of new monitoring capabilities in programmable data planes. The prototypes of P4LogLog, P4NEntropy and P4DDoS have been implemented with the P4 behavioral model [15] and proved to be fully executable in a P4 emulated environment.

We then evaluate P4LogLog, P4NEntropy and P4DDoS by means of simulations to show their effectiveness and their sensitivity to different tuning parameters, with three critical improvements (to the best of our knowledge) with respect to the literature:

- P4LogLog can guarantee better accuracy than a widely-adopted state-of-the-art flow cardinality estimator [16] while ensuring small memory usage.
- P4NEntropy ensures a comparable relative error in entropy estimation to a P4-based state-of-the-art solution [17], but it avoids the usage of pre-computed values stored in the Ternary Content-Addressable Memory (TCAM) and adopts a time-interval-based window instead of a packet-based one, which eases the switches' synchronization if additional network-wide operations should be executed.
- P4DDoS ensures slightly better performance than an existing P4-enabled entropy-based DDoS detection solution [17]. In case of some stealthy DDoS attacks, such as an internal botnet DDoS attack or a DDoS attack with spoofed source IPs, our P4DDoS outperforms the state-of-the-art solution in terms of detection accuracy. Moreover, our P4DDoS does not need any interaction with the control plane in executing the foreseen operations, whereas [17] requires that a controller properly populates the TCAM of the switch with some pre-computed values. This is why we claim that *our strategy works entirely in the data plane*.

The remainder of the paper is organized as follows. In Section 2 we report background notions. Section 3 motivates why we choose P4Log and P4Exp (see [1]) as building blocks for P4LogLog and P4NEntropy. Section 4 describes P4LogLog and P4NEntropy, while Section 5 describes P4DDoS. Sections 6 and 7 present evaluation results and comparisons with existing solutions. In Section 8 we recall the related work. Finally, Section 9 concludes the paper and discusses the future work.

2 Background

In this section we recall background concepts needed to understand the strategies proposed in the following sections.

2.1 Normalized network traffic entropy

Network traffic entropy [18] gives an indication on traffic distribution across the network. Each network switch can evaluate the traffic entropy related to the network flows that cross it in a given time interval T_{int}. Relying on the definition of *Shannon entropy* [14], network traffic entropy can be defined as $H = -\sum_{i=1}^{n} \frac{f_i}{|S|_{tot}} \log_d \frac{f_i}{|S|_{tot}}$, where f_i is the packet count of the incoming flow with *flow key* i (e.g. 5 tuple, source IP-destination IP pair, etc.), $|S|_{tot}$ is the total number of processed packets by the switch during T_{int}, n is the overall number of distinct flows and d is the base of logarithm. Traffic entropy is $H = 0$ when in T_{int} all packets $|S|_{tot}$ belong to the same flow i, while it assumes its maximum value $H = \log_d n$ when packets are uniformly distributed among the n flows. The *normalized entropy* is defined as $H_{norm} = \frac{H}{\log_d(n)}$ $(0 \leq H_{norm} \leq 1)$.

2.2 Hamming weight computation

Hamming weight represents the number of non-zero values in a string. In a binary string, the Hamming weight indicates the overall number of ones. For example, given the binary string 01101, the Hamming weight is 3. It can be computed by means of different algorithms: as part of P4LogLog, in this paper we adopt the *Counting 1-Bits* algorithm presented in [19], as it only relies on bitwise operations that are completely supported by the P4 language [20].

2.3 Sketch-based estimation of flow packet count

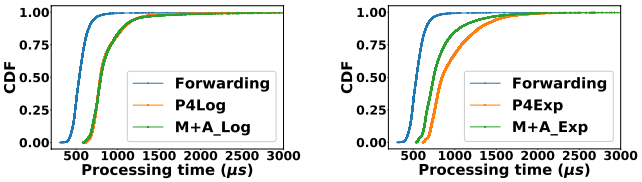
Estimating the number of packets for a specific flow crossing a programmable switch (f_i) is fundamental for network traffic entropy computation. Such an estimation can be performed by means of *sketches* [7], which are probabilistic data structures associated to a set of pairwise-independent hash functions. The *size* of each sketch data structure depends on the number of associated hash functions N_h and on the output size of each function N_s, and is $N_h \times N_s$. *Update* and *Query* operations are used to store and retrieve information from the sketch: Update operation is responsible for updating the sketch to keep track of flow packet counts, while Query operation retrieves the estimated number of packets for a specific flow. Two well-known algorithms to Update and Query sketches are *Count-min Sketch* [21] and *Count Sketch* [22]. A detailed theoretical analysis on the accuracy/memory occupation trade-off for these sketching algorithms is reported in [21][22]. From a high-level perspective, as any of N_h and N_s increase, memory consumption is larger but estimation is more accurate. Count Sketch leads to a better accuracy/memory consumption trade-off than Count-min Sketch, but its update time is twice slower [23].

2.4 LogLog algorithm for flow cardinality estimation

LogLog [13] is a sketch-based algorithm that can be adopted to estimate the number of distinct flows crossing a switch. In brief, it works as follows. Given an incoming packet with *flow key* i, LogLog applies to i a hash function with output size os: the resulted os-bit binary string s is denoted by $s = \{s_{os-1}s_{os-2}\cdots s_0\}$. LogLog then updates an m-sized LogLog register Reg. Let $bucket$ be the rightmost k bits of s (with $k = \log_2 m$) and x the remaining bits, i.e., $bucket =$

TABLE 1
P4 programs properties

Algorithm	Parameter [1]	Value [1]	Instructions	M+A entries
P4Log	N_{digits}	3	47	1
	N_{bits}	4		
P4Exp	N_{terms}	7	64	1
M+A_Log	-	-	0	1920
M+A_Exp	-	-	0	2049
Forwarding	-	-	0	1



(a) Log estimation (b) Exp estimation

Fig. 1. Cumulative distribution function of packet processing time

$\{s_{k-1} \cdots s_0\}$ and $x = \{s_{os-1} \cdots s_k\}$. Reg is updated following this rule: $Reg[bucket] = max(Reg[bucket], value)$, where $value$ is the index of the rightmost 1 of x plus one. Reg can then be queried to estimate the flow cardinality $\hat{n}$, which is computed as $\hat{n} = \alpha_m m 2^{\frac{1}{m}\sum_{bucket=0}^{m-1} Reg[bucket]}$, where α_m is a bias correction parameter. An interesting property of LogLog is that multiple LogLog sketches can be merged to a single sketch, which can be used to count the flow cardinality of the union of many packet streams.

3 Comparison of Log and Exp estimation strategies in programmable data planes

Since P4 language does not support logarithm and exponential function computations, many advanced algorithms leveraging those operations (e.g., linear counting [24]) are not directly implementable using such domain-specific language. However, these advanced algorithms are useful for executing many networking tasks in programmable data planes, including flow cardinality estimation [25] and DDoS detection [17][26], so finding a way to support them is of paramount importance. Sharma *et al.* [25] successfully implemented estimation of logarithm and exponential function in P4, but their strategy requires the storage of appropriate pre-computed values in match-action (M+A) tables. The conference version of this paper [1] proposes and evaluates the accuracy of two algorithms for exponential function and logarithm estimation, called *P4Exp* and *P4Log*, which only rely on P4-supported arithmetic operations. P4Exp and P4Log algorithms have comparable accuracy as [25] without the usage of any M+A table. However, a comparison of performance in terms of *packet processing time* between P4Exp, P4Log and the corresponding state of the art strategies is missing in [1]. We believe that such a comparison is important to understand how different approaches affect packet processing in the P4 pipeline, and to take a decision on which exponential function and logarithm estimation algorithms we should leverage for the design and implementation of P4LogLog, P4NEntropy, and P4DDoS.

We chose Mininet [27] as an emulated network environment with a single P4 switch. The data plane pipeline is described by P4 code compiled using the *bmv2* behavioral model [15]. We then connected the P4 switch to two hosts, ensuring that packets can be forwarded from one to the other. Virtual links bandwidth is bounded by CPU capacity. The emulated environment is built on top of a virtual machine deployed by OpenStack on our local testbed with dedicated access to 4 $\times$ 2.7GHz CPU cores and to 4GB of RAM. We used Wireshark [28] to capture a packet timestamp t_{in} at the ingress interface of the switch and the timestamp t_{out} at the egress interface when the same packet is forwarded to the destination host. The packet processing time is then calculated by $t_{out} - t_{in}$.

In addition to P4Exp and P4Log implementations, we implemented the logarithmic and exponential function estimation strategies reported in [25], named here as *M+A_Log* and *M+A_Exp*, respectively. For 64-bit operands in P4, to ensure a relative error below 1% for the estimated values with respect to the real ones, M+A_Exp needs 2048 entries in an exact match table, while M+A_Log requires 1919 entries in a ternary match table (to be stored in TCAM). However, in behavioral model [15], any M+A table can include at most 1024 entries, so we had to assign two exact match tables for M+A_Exp and two ternary match tables for M+A_Log. A simpler benchmark strategy, named *Forwarding*, is also implemented: it only requires a M+A table for forwarding the packets from source to destination host according to predefined flow rules. All the other strategies implement the same forwarding logic in their pipeline. P4Log and P4Exp parameters are taken from [1] and shown in Tab. 1. The table also summarizes the number of required instructions (i.e., logical and arithmetical operations) and of M+A entries for all the considered strategies (including forwarding capabilities), showing the inherent differences of the approaches.

We then evaluate the packet processing time, considering base-2 logarithm and exponential function. We generated and forwarded 10000 packets: Fig. 1 shows the cumulative distribution function (CDF) of packet processing time. No packet loss was experienced. As shown in Fig. 1(a), both P4Log and M+A_Log cause a higher processing time than Forwarding since they need to carry out more complex operations. However, their CDF curves are almost overlapped: this means that P4Log does not cause any additional overhead on processing time with respect to M+A_Log, but it has the benefits of not requiring any M+A table. Likewise, Fig. 1(b) reveals that both exponential function estimation strategies slightly increase the processing time with respect to Forwarding. P4Exp has just a slightly higher packet processing time than M+A_Exp but, also in this case, it does not require any M+A table to work. Note also that packets, in real high-performance programmable switches, are expected to be processed in few hundreds of *ns* [29], thus such a difference in processing time would impact even less on performance (in absolute terms).

We have shown that P4Log and P4Exp have comparable accuracy (see [1]) and efficiency as the state of the art, while preventing from the usage of expensive and power-hungry switch memory (e.g. TCAM) for their execution: we thus chose to leverage P4Log and P4Exp for all the logarithmic and exponential-function estimations needed in the following Sections. P4Log and P4Exp can then be seen as two primitives, while the reader interested in their

implementation should refer to [1].

As a final remark, we want to remind that, according to [1], the output of P4Log(x) is $\log_2 x \ll 10$ (i.e., $\log_2 x$ left-shifted 10 bits), while the the output of P4Exp(x) is 2^x. 10-bit left-shifting operation $\ll$ (i.e., multiplying by 2^{10}) is done in [1] to "amplify" decimal numbers and maintain the information carried by their decimal part (which would be truncated by P4 otherwise). 10-bit right-shifting $\gg$ can be instead used to move back an amplified number to its original range. Note that P4Exp can be used for the exponential function computation of any real positive number, while in the case of natural numbers it is more efficient to exploit the left shift operator (i.e., $2^x \leftarrow 1 \ll x$ for x integer positive). We will largely leverage such properties and logical operations in the next sections of this paper.

4 Estimation of Normalized Traffic Entropy

Based on P4Log and P4Exp we initially propose *P4LogLog* to estimate flow cardinality. P4LogLog is then used by *P4NEntropy*, which estimates the normalized network traffic entropy. The prototypes of both strategies have been implemented in P4 behavioral model [15] and are executable in an emulated environment as Mininet [27]. The P4 source codes are available in [30] and [31].

4.1 Flow cardinality estimation: P4LogLog

In this section we propose P4LogLog for the estimation of flow cardinality. The problem is formulated as follows.

Problem definition: *Given* a stream S of incoming packets, each one belonging to a specific *flow* i, *returns* the estimated *flow cardinality* $\hat{n}$ of S, i.e., the estimated number of distinct flows in S.

For instance, if we identify as *flow key* i each packet *destination IP*, meaning that a flow includes all the packets towards a specific destination, then the flow cardinality of destination IPs represents the number of destination IPs in the network. Same consideration holds for any other flow definition (e.g. packets with the same 5-tuple, same source/destination IP pair, etc.) without any loss of generality. In the following, we report the details of Update and Query operations of P4LogLog, which both follow specifications from LogLog [13] (see Section 2.4) while only using P4-supported instructions.

4.1.1 Update

As shown in Algorithm 1, *Update* function iteratively updates a readable and writable stateful register Reg for each incoming packet, which belongs to a flow with flow key i. The flow key i of the packet is hashed by a given hash function, and the output value is converted to a os-bit binary string s (Line 6). In this paper, we consider $os = 32$ and an m-sized register Reg, where $m = 2^k$ and integer $k \in \{4, ..., 16\}$ (as per [13]). The index of the register's cell to be updated, named $bucket$ ($0 \leq bucket \leq m-1$), is the binary number represented by the rightmost k bits of s, which can be obtained by $s\&(2^k-1)$, i.e., $s\&0\underbrace{11\cdots1}_{k}$ (Line 7). & is the bitwise inclusive AND operator and $2^k - 1$ (in binary) is pre-stored in the P4 program once k is chosen. The algorithm then right-shifts s to k bits to get a binary string x where the first k bits are 0s and the remaining $os-k$ bits are the first $os-k$ bits of s (Line 8). The index of rightmost 1 in x, called *value*, is then used to update the LogLog register's cell in $bucket$ position. Unfortunately, retrieving such rightmost 1 is not trivial. As shown from Lines 9 to 12, the algorithm adopts the following strategy: all bits of x on the left of the rightmost 1 are iteratively converted to 1, and the result of this iterative operation is stored in w (| is the bitwise inclusive OR operator). For example, an os-bit binary value $x = \underbrace{00\cdots01}_{os-1}0$ is converted to $w = \underbrace{11\cdots11}_{os-1}0$. The algorithm for *Hamming weight* recalled in Section 2.2 is then used to count b, i.e., the number of 1s in x (Line 12): $value$ is equal to $os+1-b$ (Line 13). Finally, if $value$ is larger than the $bucket$-indexed value in the register, $value$ replaces the stored value (Lines 14-15).

Algorithm 1: P4LogLog

```
   Input: Packet stream S
   Output: Flow cardinality estimation n̂
1  m ← 2^k (k ∈ {4, ..., 16})
2  os ← 32
3  Reg ← m-sized empty LogLog register
4  Function Update(Reg):
5      for Each received packet belonging to flow i do
6          s ← (Hash(i) → {0,1}^os)
7          bucket ← s&(2^k − 1)
8          x ← (s ≫ k)
9          w ← x|(x ≪ 1)
10         for int l ∈ {1, ··· , log2(os) − 1} do
11             w ← w|(w ≪ 2^l)
12         b ← HammingWeight(w)
13         value ← os + 1 − b
14         if value > Reg[bucket] then
15             Reg[bucket] ← value
16     return Reg
17 α_m ← 0.39701 ≪ 10
18 Function Query(Reg):
19     exp ← P4Exp((Σ_{bucket=0}^{m−1} Reg[bucket]) ≫ k)
20     n̂ ← (exp · α_m · m) ≫ 10
21     return n̂
```

4.1.2 Query

Query function in Algorithm 1 estimates the flow cardinality directly in the switch. The flow cardinality estimation $\hat{n}$ is computed as in [13] and Section 2.4 from all LogLog register's stored values by exploiting P4Exp. The k-bit right-shift operation carried out on the sum of values from Reg is equivalent to dividing such sum by $m = 2^k$ (Line 19). The floating parameter α_m, chosen as in [13], is amplified 2^{10} times through left shift operation, and the resulted value from the computation executed in Line 20 is right-shifted 10 bits to get the estimated flow cardinality $\hat{n}$.

4.2 Normalized traffic entropy estimation: P4NEntropy

In this section we present a new strategy, named *P4NEntropy*, to estimate the normalized network traffic entropy in a given time interval using the P4 language. Formally, the problem is defined as follows.

Problem definition: *Given* a stream S of incoming packets, each one belonging to a specific *flow* i, and a time interval T_{int}, *returns* the *normalized Shannon entropy* estimation H_{norm} (see Section 2.1) at the end of T_{int}.

4.2.1 Derivation of estimated normalized entropy in P4

The goal of this section is to provide an estimation of network traffic normalized entropy by only using P4-supported operations and reducing as much as possible their number. The section also shows how relevant statistics, used for normalized entropy estimation at the end of T_{int}, are iteratively updated every time a packet crosses the switch.

We first rewrite the Shannon entropy as follows:

$$H(|S|_{tot}) = -\sum_{i=1}^{n} \frac{f_i(|S|_{tot})}{|S|_{tot}} \log_d \frac{f_i(|S|_{tot})}{|S|_{tot}}$$
$$= \log_d |S|_{tot} - \frac{1}{|S|_{tot}} \sum_{i=1}^{n} f_i(|S|_{tot}) \log_d f_i(|S|_{tot})$$

We consider $d = 2$ without any loss of generality. With respect to the definition given in Section 2.1, we use the notation $f_i(|S|_{tot})$ to make explicit that f_i refers to its value when $|S|_{tot}$ packets have been received (i.e., at the end of T_{int}). As packets arrive to the switch, the overall number of processed packets $|S|$ increases and must be stored in the switch to ensure that $H(|S|_{tot})$ can be computed at the end of T_{int}, when $|S| = |S|_{tot}$. We define $Sum(|S|) = \sum_{i=1}^{n} f_i(|S|) \log_d f_i(|S|)$, which must be updated as well. To understand how to update $Sum(|S|)$, let's assume that a new packet for a specific flow arrives and it is the $|S|$-th packet. We call its packet count $\bar{f}_i(|S|)$. It holds that:

$$\begin{cases} f_i(|S|) = f_i(|S|-1) & (f_i(|S|) \neq \bar{f}_i(|S|)) \\ f_i(|S|) = f_i(|S|-1) + 1 & (f_i(|S|) = \bar{f}_i(|S|)) \end{cases}$$

This allows us to re-write $Sum(|S|)$ as follows:

$$Sum(|S|) = Sum(|S|-1) + \bar{f}_i(|S|) \log_2 \bar{f}_i(|S|) + - (\bar{f}_i(|S|) - 1) \log_2 (\bar{f}_i(|S|) - 1)$$

$Sum(|S|)$ thus needs two logarithmic computations for each incoming packet, and would require running P4Log twice with corresponding computational effort.

In the next step, we show how it is possible to estimate $Sum(|S|)$ with only (at most) one logarithmic computation. When $\bar{f}_i(|S|) = 1$, we estimate $Sum(|S|) = Sum(|S| - 1)$, being $\bar{f}_i(|S|) \log_2 \bar{f}_i(|S|) = 1 \log_2 1 = 0$ and defining $(\bar{f}_i(|S|) - 1) \log_2 (\bar{f}_i(|S|) - 1) = 0 \log_2 0 = 0$ [6]. Instead, when $\bar{f}_i(|S|) > 1$, we need to re-write once again $Sum(|S|)$ in the following way:

$$Sum(|S|) = Sum(|S|-1) + \log_2 \bar{f}_i(|S|) + + (\bar{f}_i(|S|) - 1) \log_2 (1 + \frac{1}{\bar{f}_i(|S|) - 1})$$

According to L'Hopital's rule [32]:

$$\lim_{\bar{f}_i(|S|) \to +\infty} (\bar{f}_i(|S|-1) \log_2 (1 + \frac{1}{(\bar{f}_i(|S|-1)}) = \frac{1}{\ln 2}$$

Thus, we set $1/\ln 2 \approx 1.44$ as the approximation of the third term of $Sum(|S|)$. This approximation best works when most of the flows in T_{int} carry a number of packets much greater than 1 (as it usually happens in an ISP backbone network, which is the most suitable scenario where to apply our strategy). Finally, $Sum(|S|)$ can be estimated as:

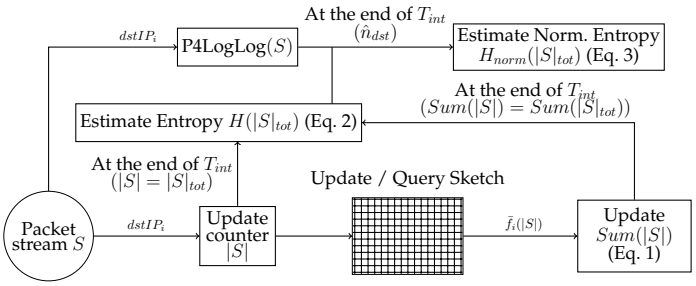


Fig. 2. Scheme of P4NEntropy

$$Sum(|S|) \approx \begin{cases} Sum(|S|-1) & (\bar{f}_i(|S|) = 1) \\ Sum(|S|-1) + \log_2 \bar{f}_i(|S|) + 1/\ln 2 & \\ & (\bar{f}_i(|S|) > 1) \end{cases} \quad (1)$$

This estimation requires at most one logarithm computation. Since P4 language does not support division, we re-write $\frac{1}{|S|_{tot}} = 2^{-\log_2 |S|_{tot}}$. So, entropy can be written as:

$$H(|S|_{tot}) = \log_2 |S|_{tot} - 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})}$$

In this form, entropy can be estimated by only using P4-supported operations, leveraging P4Log and P4Exp algorithms. In the following we show how, in some cases, it is possible to further slightly reduce complexity in entropy estimation. When $|S|_{tot} = \sum_{i=1}^{n} f_i(|S|_{tot}) > Sum(f_i|S_{tot}|)$, it holds that $0 < 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})} < 1$. This is a corner case that happens only when flow distribution is almost uniform (i.e., when most of flows carry only one or very few packets). In this case, we neglect the computation of $2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})}$, meaning that we estimate entropy as flow distribution was perfectly uniform. Network traffic entropy can then be estimated as follows:

$$H(|S|_{tot}) \approx \begin{cases} \log_2(|S|_{tot}) & (|S|_{tot} > Sum(|S|_{tot})) \\ \log_2(|S|_{tot}) - 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})} & \\ & (|S|_{tot} \leq Sum(|S|_{tot})) \end{cases} \quad (2)$$

Finally, normalized entropy $H_{norm}(|S|_{tot})$ is estimated as:

$$H_{norm}(|S|_{tot}) = 2^{\log_2(H(|S|_{tot})) - \log_2(\log_2 \hat{n})} \quad (3)$$

The number of estimated distinct flows $\hat{n}$ can be obtained using P4LogLog, that is, by updating a LogLog register for each incoming packet and by querying it at the end of T_{int}.

4.2.2 Description of P4NEntropy strategy

Figure 2 and Algorithm 2 show the scheme and pseudocode of P4NEntropy algorithm, leveraging outcomes from Sections 4.1 and 4.2.1. First, the algorithm continuously updates $Sum(|S|)$ until the end of T_{int} (*UpdateSum* function) with flow information from incoming packets. A counter $|S|$ is used to count all incoming packets in the switch. Note that we consider as *flow key* the destination IP of the packet, with $i \sim dstIP_i$. A sketch data structure (e.g., Count Sketch or Count-min Sketch, see Section 2.3) is used to store the estimated packet count for all the flows, being continuously updated to include information from new packets, and then it is queried to retrieve the estimated packet count $\bar{f}_i(|S|)$ for the flow i the current incoming packet belongs to. This value is then passed to a register named $Sum(|S|)$, which is updated as specified in Eq. 1. All the floating numbers

Algorithm 2: P4NEntropy

Input: Packet stream S, time interval T_{int}
Output: Normalized entropy estimation $H_{norm}(|S|_{tot})$ of S in T_{int}

1 $|S| \leftarrow 0$, $Sum(|S|) \leftarrow 0$
2 **Function** *UpdateSum*($Sum(|S|)$)**:**
3 **while** *currentTime* $< T_{int}$ **do**
4 **for** *Each received packet belonging to flow* i **do**
5 $|S| \leftarrow |S| + 1$
6 $\bar{f}_i(|S|) \leftarrow Sketch(dstIP_i)$
7 **if** $\bar{f}_i(|S|) > 1$ **then**
8 $Sum(|S|) \ll 10 \leftarrow Sum(|S|) \ll 10$
9 $+P4Log(\bar{f}_i(|S|)) + 1.44 \ll 10$
10 $Sum(|S|_{tot}) \leftarrow (Sum(|S_{tot}|) \ll 10) \gg 10$
11 **return** $Sum(|S|_{tot})$, $|S|_{tot}$
12 **Function** *EstimateNormEntropy*($Sum(|S|_{tot})$, $|S|_{tot}$)**:**
13 **if** *currentTime* = T_{int} **then**
14 **if** $|S|_{tot} > Sum(|S|_{tot})$ **then**
15 $H(|S|_{tot}) \ll 10 \leftarrow P4Log(|S|_{tot})$
16 **else**
17 diff $\leftarrow P4Log(Sum(|S|_{tot})) - P4Log(|S|_{tot})$
18 $H(|S|_{tot}) \ll 10 \leftarrow P4Log(|S|_{tot}) - P4Exp(2, \text{diff})$
19 $\hat{n}_{dst} \leftarrow P4LogLog(S, T_{int})$
20 $\text{diff}_n \leftarrow P4Log(H(|S|_{tot}) \ll 10) +$
21 $-(P4Log(P4Log(\hat{n}_{dst})) - 10 \ll 10)$
22 **if** $diff_n > 0$ **then**
23 $H_{norm}(|S|_{tot}) \ll 10 \leftarrow P4Exp(2, \text{diff}_n)$
24 **else**
25 $H_{norm}(|S|_{tot}) \ll 10 \leftarrow 0$
26 **return** $H_{norm}(|S|_{tot}) \ll 10$

in the equation must be amplified 2^{10} times, since P4Log outputs an amplified integer value. Only at the end of T_{int}, $Sum(|S|_{tot})$ is reduced by a factor of 2^{10} and its final value, together with $|S|_{tot}$, is returned (Lines 1-11 of the pseudocode). Traffic entropy is then estimated as specified in Eq. 2 (Lines 13-18). The resulted value of $H(|S|_{tot})$ is amplified 2^{10} times since output values of P4Log are amplified, while output values of P4Exp are not. Such an amplification makes it possible to use P4Exp in Eq. 3 to estimate $H_{norm}(|S|_{tot})$ amplified 2^{10} times. Note that $H(|S|_{tot}) \ll 10$ may be smaller than $\log_2(\hat{n}_{dst})$ but, in this case, the normalized network traffic entropy can be approximated to 0 (Line 25). Since the result of P4Log is left-shifted 10 bits, the computation of $\log_2(\log_2(\hat{n}_{dst}))$ must be carefully handled. Considering that the result of P4Log($\hat{n}_{dst}$) is $\log_2(\hat{n}_{dst}) \ll 10$, the output of P4Log($\log_2(\hat{n}_{dst}) \ll 10$) can be expressed as $\log_2(\log_2(\hat{n}_{dst}) \ll 10) \ll 10 = \log_2(\log_2(\hat{n}_{dst}) \cdot 2^{10}) \ll 10 = \log_2(\log_2(\hat{n}_{dst})) \ll 10 + 10 \ll 10$. Hence, $\log_2(\log_2(\hat{n}_{dst})) \ll 10$ is equivalent to P4Log(P4Log($\hat{n}_{dst}$)) $-10 \ll 10$ (Line 21). The resulting value is used to compute the normalized network traffic entropy amplified 2^{10} times (Line 23).

5 Entropy-based DDoS detection

Based on P4NEntropy, we present a simple yet effective entropy-based DDoS detection strategy in P4, named *P4DDoS*. The P4 code of P4DDoS is available in [33]. Formally, the problem is defined as follows.

Problem definition: *Given* a k-th time interval T_{int}^k, a stream S_k of incoming packets during T_{int}^k, the estimated normalized network traffic entropy of destination IPs H_{norm}^k at the end of T_{int}^k and an adaptive threshold λ_{norm}^k, *returns* an *alarm* to the controller, at the end of T_{int}^k, if a *potential DDoS attack* is identified.

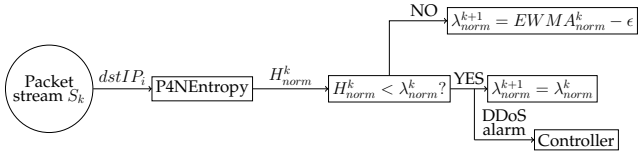


Fig. 3. Scheme of P4DDoS

Our proposed strategy triggers an alarm (e.g. a flag embedded in a field of the *report packet* header) if $H_{norm}^k < \lambda_{norm}^k$. In fact, as empirically evaluated in previous works (e.g. [34][35]), when a DDoS attack occurs, the normalized network traffic entropy of destination IPs significantly decreases, since traffic is concentrated around few destination nodes. The most critical aspect for such an entropy-based strategy is how to set the threshold λ_{norm}^k. This will be discussed in the next subsection. Note also that we only focus on *volumetric* DDoS attacks (e.g. UDP flooding or DNS amplification attacks); considering other types of attacks, such as *link flooding* or *carpet bombing*, is left as future work.

5.1 Adaptive threshold setting

Since network traffic fluctuates over time, we define an *adaptive threshold* to protect our strategy from false positives that may be generated if using a fixed-value threshold in such a dynamic environment. Our proposed adaptive threshold leverages the computation of an *Exponentially Weighted Moving Average* (EWMA) of H_{norm}^k across different time intervals. The moving average $EWMA_{norm}^k$ in time interval T_{int}^k is expressed as:

$$EWMA_{norm}^k = \begin{cases} H_{norm}^k & (k = 1) \\ \alpha H_{norm}^k + (1 - \alpha) EWMA_{norm}^{k-1} & (k > 1) \end{cases}$$

where α $(0 < \alpha < 1)$ is the smoothing factor for $EWMA_{norm}^k$. We define a threshold parameter ϵ $(0 \leq \epsilon \leq 1)$, used to compute the threshold λ_{norm}^{k+1} in the next time interval T_{int}^{k+1} if no alarm is generated in T_{int}^k:

$$\lambda_{norm}^{k+1} = \begin{cases} EWMA_{norm}^k - \epsilon & \text{(no alarm in } T_{int}^k) \\ \lambda_{norm}^k & \text{(alarm in } T_{int}^k) \end{cases}$$

As shown above, the threshold λ_{norm}^{k+1} is not updated if an alarm is generated in the time interval: this ensures that the threshold is updated when only legitimate traffic crosses the switch and its value is not biased by DDoS traffic. Note that setting the parameter ϵ in a proper way is also fundamental to get good DDoS detection performance. This aspect will be evaluated in Section 7.

5.2 Implementation in P4 language

Figure 3 and Algorithm 3 report the scheme and pseudocode of the P4DDoS strategy, with focus on a given time interval T_{int}^k. At the end of time interval T_{int}^k, the *DDoS-Detection* function is executed. $Alarm_{ddos}^k$ is set to 0 and the normalized network traffic entropy H_{norm}^k is estimated by P4NEntropy, amplified 2^{10} times (Lines 2-3). It is then compared to the threshold $\lambda_{norm}^k \ll 10$ (Line 4). If smaller, the alarm $Alarm_{ddos}^k$ is set to 1 and the *UpdateThreshold*

Algorithm 3: P4DDoS

Input: Packet stream S_k, time interval T_{int}^k, threshold parameter ϵ, smoothing factor α, threshold $\lambda_{norm}^k \ll 10$ and average $EWMA_{norm}^{k-1} \ll 10$ computed in T_{int}^{k-1}
Output: DDoS alarm $Alarm_{ddos}^k = 1$ if a DDoS attack is detected in T_{int}^k

1 **Function** *DDoSDetection*($\lambda_{norm}^k \ll 10$)**:**
2 $H_{norm}^k \ll 10 \leftarrow$ P4NEntropy(S_k, T_{int}^k)
3 $Alarm_{ddos}^k \leftarrow 0$
4 **if** $H_{norm}^k \ll 10 < \lambda_{norm}^k \ll 10$ **then**
5 $Alarm_{ddos}^k \leftarrow 1$
6 *UpdateThreshold*($k, \alpha, H_{norm}^k \ll 10, \epsilon, Alarm_{ddos}^k$,
7 $EWMA_{norm}^{k-1} \ll 10, \lambda_{norm}^k \ll 10$)
8 **else**
9 *UpdateThreshold*($k, \alpha, H_{norm}^k \ll 10, \epsilon, Alarm_{ddos}^k$,
10 $EWMA_{norm}^{k-1} \ll 10, \lambda_{norm}^k \ll 10$)
11 **return** $Alarm_{ddos}^k$
12 **Function** *UpdateThreshold*($k, \alpha, H_{norm}^k \ll 10, \epsilon$,
13 $Alarm_{ddos}^k, EWMA_{norm}^{k-1} \ll 10, \lambda_{norm}^k \ll 10$)**:**
14 **if** $Alarm_{ddos}^k = 0$ **then**
15 **if** $k = 1$ **then**
16 $EWMA_{norm}^k \ll 10 \leftarrow H_{norm}^k \ll 10$
17 **else**
18 $EWMA_{norm}^k \ll 10 \leftarrow ((\alpha \ll 10) \cdot H_{norm}^k \ll 10$
19 $+((1-\alpha) \ll 10) \cdot EWMA_{norm}^{k-1} \ll 10) \gg 10$
20 $\lambda_{norm}^{k+1} \ll 10 \leftarrow EWMA_{norm}^k \ll 10 - \epsilon \ll 10$
21 **else**
22 $\lambda_{norm}^{k+1} \ll 10 \leftarrow \lambda_{norm}^k \ll 10$
23 **return** $\lambda_{norm}^{k+1} \ll 10, EWMA_{norm}^k \ll 10$

function is called (Lines 5-7). Otherwise, the *UpdateThreshold* function is called without changing $Alarm_{ddos}^k$ (Lines 9-10). If $Alarm_{ddos}^k = 1$, the switch clones the current packet and embeds the value 1 in a customized header field. This report packet is then sent to the controller to report that a *potential* DDoS attack has been detected. It is possible to embed more information in the header of the report packet, such as the estimated network traffic entropy. In this case, the controller is able to take a network-wide decision using the entropy retrieved from multiple switches (see Section 5.3.1) about whether the *potential* DDoS attack is an *actual* attack.

The *UpdateThreshold* function updates EMWA and the adaptive threshold as specified in Section 5.1. (Lines 12-23). Note that, since both EMWA and the threshold λ_{norm} are usually decimal numbers, all the operations are executed to ensure that their value is amplified 2^{10} times.

5.3 Insights and discussions

5.3.1 Network-wide coordination

So far, we have focused on entropy-based DDoS detection in a single programmable switch. The switch can generate alarms if, according to the traffic flowing through its interfaces, a DDoS attack may be occurring. However, given the reduced network visibility of a single switch, a final decision on whether a DDoS attack is actually carried out should be taken by the centralized controller from a *network-wide* perspective, that is, by cross-checking collected information from multiple switches and taking a global decision. For instance, UnivMon [6] and Elastic Sketch [16] present a way to estimate *network-wide traffic entropy*: the idea behind those works is to sample a set of flows with large packet count in any programmable switch, and send such statistics to the controller at the end of any time interval. The controller estimates the entropy of reported sampled "heavy" flows and considers it as a *network-wide entropy estimation*. As reported in [36], these two approaches assume that packets for a specific flow are counted only once in the network. By making the same strong assumption, in our case network-wide traffic entropy H^{nw} can be expressed as:

$$H^{nw} = \log_2(\sum_{j=1}^{w} |S_j|_{tot}) - \frac{1}{\sum_{j=1}^{w} |S_j|_{tot}} (\sum_{j=1}^{w} Sum_j)$$

where w is the number of switches in the network and $Sum_j = \sum_{i=1}^{n_j} f_i(|S_j|_{tot}) \log_d f_i(|S_j|_{tot})$ (see Section 4.2.1). Additionally, according to the union property of LogLog (see section 2.4), the normalized network-wide traffic entropy H_{norm}^{nw} can be expressed as:

$$H_{norm}^{nw} = \frac{H^{nw}}{\log_2(LogLog(S_1 \cup S_2 \cup \cdots S_w))}$$

In this latter case, the strong assumption above can be neglected, since the union property of LogLog makes it possible to estimate the network-wide number of distinct flows also if a packet is counted in different locations.

Given the above considerations, a network-wide strategy could be designed to forward to the controller all the needed information from the switches (i.e., $|S_j|_{tot}$, Sum_j and the j-th LogLog register) for the computation of network-wide normalized entropy in support to a centralized *network-wide DDoS detection*. However, since in real scenarios the packet may traverse multiple switches and generate duplicated packet counts, the accuracy of the computed network-wide entropy N^{nw} would be compromised. How to overcome this issue is still open: we will work on refined strategies for network-wide entropy-based DDoS detection in the future.

5.3.2 Implementation in a programmable hardware switch

We tried to implement P4DDoS in a P4-programmable hardware switch with Tofino Application-Specific Integrated Circuit [37]. Due to limited hardware resources, we could not fully implement it. However, hardware vendors are currently launching more and more powerful P4-programmable switches, so we are pretty confident that in the future it will be possible to execute P4DDoS in hardware targets, while ensuring a line-rate packet processing speed with only a few-hundreds of nanoseconds packet processing latency, as already possible for simpler strategies [29].

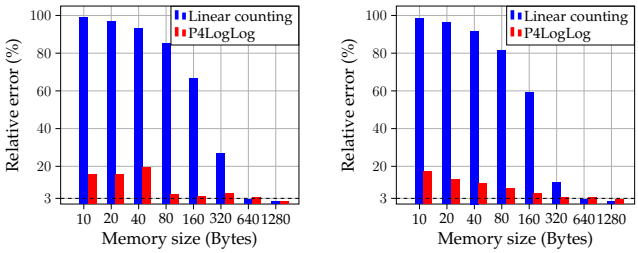
6 P4LogLog and P4NEntropy evaluation

We implemented P4LogLog and P4NEntropy in Python and simulated them for evaluation.

6.1 Evaluation metrics and simulation settings

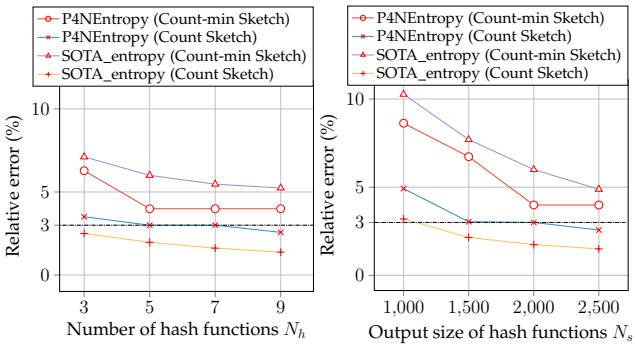
6.1.1 Testing flow trace and methodology

P4LogLog: We use 2018-passive CAIDA flow trace [38], including 50 seconds of network traffic, and divide it into 50 1-second time intervals (or observation windows). In each considered time interval there are around 460K packets.



(a) Flow key: source IP (b) Flow key: destination IP

Fig. 4. Performance comparison of P4LogLog with an existing flow cardinality estimation approach [5]



(a) Sensitivity to N_h (b) Sensitivity to N_s

Fig. 5. Performance comparison of P4NEntropy with an existing entropy estimation approach [17] (flow key: destination IP)

P4NEntropy: We use the same CAIDA flow trace but divide it into 10 observation windows of roughly 5 seconds each, every one including a fixed number of 2^{21} packets. Fixing the number of packets per observation window is needed to compare our approach with a state-of-the-art solution [17], named *SOTA_entropy* for the remainder of the section, which adopts M+A tables to store pre-computed values and only works with windows with power-of-two number of packets.

6.1.2 Evaluated metrics

We consider *relative error* as an evaluation metric.

P4LogLog: Being n the exact number of distinct flows (either identified by *source IP* or *destination IP* as flow key) in a time interval and $\hat{n}$ its estimated value, the relative error is defined as the average value of $\frac{|n-\hat{n}|}{n} \cdot 100\%$ in all the consecutive 50 time intervals.

P4NEntropy: We call $\hat{H}$ the estimated traffic entropy of *destination IPs* in an observation window and H its exact value. The relative error is defined the average value of $\frac{|H-\hat{H}|}{H} \cdot 100\%$ in the 10 consecutive observation windows. Note that we evaluate the entropy H and not its normalized value H_{norm}: this is needed to make a fair comparison with SOTA_entropy, which does not consider any entropy normalization. To understand how normalization affects the accuracy of the estimated entropy, the reader should refer to the evaluation of P4LogLog (Section 6.2).

6.1.3 Tuning parameters

The default tuning parameters for P4Log and P4Exp, adopted for both P4LogLog and P4NEntropy, are set as in Tab. 1. The sketch (either Count-min or Count Sketch) used by P4NEntropy has default size $(N_h = 5) \times (N_s = 2000)$.

6.2 Evaluation of P4LogLog

As shown in Fig. 4, we compare our P4LogLog with another existing flow cardinality estimator (*Linear counting* [5]), implementable in a programmable data plane, in terms of relative error. 1 bit is used for each Linear counting register cell [39], while 5 bits are allocated for each P4LogLog register cell [13]. Given this, we vary the *memory size* of each register for both the approaches (i.e., we vary the number of cells in the registers, which can be easily retrieved).

Figure 4(a) focuses on the estimation of distinct source IPs in the trace. The relative error on such a flow cardinality estimation by adopting Linear counting is 50% higher than by adopting P4LogLog when the memory size is below 320 bytes, and its value for Linear counting is high for any memory size below 640 bytes. Conversely, our P4LogLog leads to acceptable relative errors with only 80 bytes. If we assign 1280-bytes registers to P4LogLog and Linear counting, the relative error of both is around 1%. Likewise, Fig. 4(b) shows the estimated number of distinct destination IPs in the trace. Our P4LogLog algorithm still outperforms Linear counting for small memory sizes. When the memory occupation reaches 640 bytes, the relative error of P4LogLog is below 3%, which is assumed as an acceptable target.

Another solution for flow cardinality estimation is proposed in [6]. However, such a solution always needs much more memory than Linear Counting and P4LogLog (i.e., at least 0.2MB) to get reasonable accuracy.

6.3 Evaluation of P4NEntropy

We simulate both our strategy and SOTA_entropy in the case that the packet count of each flow (identified by destination IP flow key) is estimated in the data plane by adopting either Count-min Sketch or Count Sketch (see Section 2.3 and Fig. 2).

We show how entropy estimation of destination IPs is affected while changing the size $N_h \times N_s$ of the sketch (Fig. 5). Fig. 5(a) shows the relative error in entropy estimation for the two strategies when N_s is fixed and N_h varies. It shows that the relative error slightly decreases as N_h increases in all the cases. Moreover, P4NEntropy and SOTA_entropy lead to similar relative error. It can be noted that, when adopting Count-min Sketch, both P4NEntropy and SOTA_entropy have larger relative error (always above 4%) than when adopting Count-Sketch. Additionally, in this case, the relative error of SOTA_entropy is slightly higher than the one of P4NEntropy, which is caused by the different ways how $Sum(f_i)$ is estimated. In SOTA_entropy, the Longest Prefix Match (LPM) lookup table for $F(f_i) = f_i \log_2 f_i - (f_i - 1)\log_2(f_i - 1)$ (see [17]) is sensitive to the large packet count (f_i) overestimation caused by Count-min Sketch. Conversely, P4NEntropy needs to calculate $\log_2 f_i + \frac{1}{\ln 2}$ (see Eq. 1), which is less sensitive to large overestimations *(i)* due to the logarithm nature and *(ii)* because $\frac{1}{\ln 2}$ is a constant value. This effect does not happen when Count-Sketch is adopted, since overestimations are much less frequent. In that case, P4NEntropy leads to slightly worse results than SOTA_entropy because, unlike SOTA_entropy, it uses an approximation for the computation of entropy (see Eq. 2).

Fig. 5(b) shows instead the impact of a variation of N_s on relative error in entropy estimation. Results are similar to what shown in Fig. 5(a), but it can be noted that both strategies are more sensitive to a variation of N_s than of N_h. In this case, when adopting Count Sketch, relative error is always close to 3%. Note that a relative error of 3% is the maximum possible value ensuring that accuracy of practical monitoring applications is not affected [18].

7 P4DDoS EVALUATION

We implemented P4DDoS in Python and simulated it for evaluation. Additionally, we also implemented a state-of-the-art entropy-based DDoS detection approach [17] executable in programmable switches, named *SOTA_DDoS* for the sake of brevity, and compared them. To make a fair comparison, both DDoS detection strategies have been implemented leveraging our proposed P4NEntropy strategy and using a sketch, for packet count estimation, of the same size. Note, however, that the original version of SOTA_DDoS uses SOTA_Entropy for entropy estimation (see the previous subsection). Unlike P4DDoS, which triggers a DDoS alarm only when the *normalized* entropy of destination IPs decreses below a threshold, SOTA_DDoS triggers a DDoS alarm when any of two conditions holds: *(i)* entropy (not normalized) of source IPs increases above an adaptive threshold and *(ii)* entropy (not normalized) of destination IPs decreases below an adaptive threshold.

7.1 Evaluation metrics and simulation settings

7.1.1 Testing flow trace and methodology

We consider three kinds of flow traces.

***Trace1*. Legitimate flow trace:** The same CAIDA flow trace [38] that we used for the evaluation of P4LogLog. The 50-seconds flow trace is divided into 50 1-second time intervals.

***Trace2*. Legitimate flow trace mixed with Booter DDoS attack traffic [40]:** 50-seconds traces taken from a set of Booter DDoS attack traces, and split into 50 time intervals. Each 1-second attack trace is injected into the legitimate 50-seconds flow trace according to its sequential 1-second time intervals. We took four different packet-rate Booter DDoS attack traces into consideration: Tab. 2 reports their properties and names as specified in [40]. We considered the four traces with the highest number of attack source IPs: this allows us to analyze DDoS attacks with different volumes. Moreover, we also injected all four Booter DDoS attack traces together into the legitimate flow trace: we name this trace as *Mixed*. Such a mixed DDoS attack flow trace can help us evaluate the performance of DDoS detection when multiple DDoS attacks occur simultaneously in the network.

***Trace3*. Legitimate flow trace mixed with internal Botnet DDoS attack traffic:** In this case, we assume that some internal hosts of the network (e.g., a datacenter network) are exploited by an attacker to reverse malicious traffic towards a DDoS victim within the same network. We varied the *attack traffic proportion* (i.e., the percentage of generated malicious traffic over the total traffic in the network) from 5% to 30%. This flow trace is generated by crafting Trace 1 in such a way that part of the traffic is forwarded to one specific DDoS victim (by changing the destination IP of a given proportion of the packets).

TABLE 2
Properties of DDoS flow traces [40]

DDoS trace name	Packet per second	Attack source IPs
Booter 6	$\sim$ 90000	7379
Booter 7	$\sim$ 41000	6075
Booter 1	$\sim$ 96000	4486
Booter 4	$\sim$ 80000	2970

7.1.2 Evaluation metrics

We consider *true-positive rate* D_{tp}, *false-positive rate* D_{fp} and *detection accuracy* D_{acc} as evaluation metrics. Considering that *(i)* True Positive (TP) is the number of time intervals with a triggered DDoS alarm while a DDoS attack is occurring in those intervals, *(ii)* True Negative (TN) is the number of time intervals without any triggered DDoS alarm while no DDoS attack is occurring, *(iii)* False Positive (FP) is the number of time intervals with a triggered DDoS alarm while no DDoS attack is occurring, and *(iv)* False Negative (FN) is the number of time intervals without any triggered DDoS alarm while a DDoS attack is instead occurring, the metrics introduced above are defined as:

$$D_{tp} = \frac{TP}{TP+FN} \times 100\%$$
$$D_{fp} = \frac{FP}{TN+FP} \times 100\%$$
$$D_{acc} = \frac{TP+TN}{TP+TN+FP+FN} \times 100\%$$

7.1.3 Tuning parameters

The smoothing factor in $EWMA_{norm}$ and for the thresholds defined in SOTA_DDoS is set to $\alpha = 0.13$: with this value, all the previous computed averages (up to all the 50 time intervals) have some impact on EWMA. All the parameters for P4Log and P4Exp are the ones reported in Tab. 1. We choose Count Sketch as sketch for P4NEntropy, with $(N_h = 5) \times (N_s = 2000)$. The register size in P4LogLog is set to $m = 2048$, which corresponds to 1280 Bytes of memory. The considered time intervals T_{int}, as already said, are 1-second wide. With longer T_{int}, N_h and N_s should be properly increased to ensure good entropy estimation accuracy. Finally, the normalized entropy parameter is set to $\epsilon = 0.01$ unless otherwise specified.

7.2 Detection performance (Booter DDoS attacks)

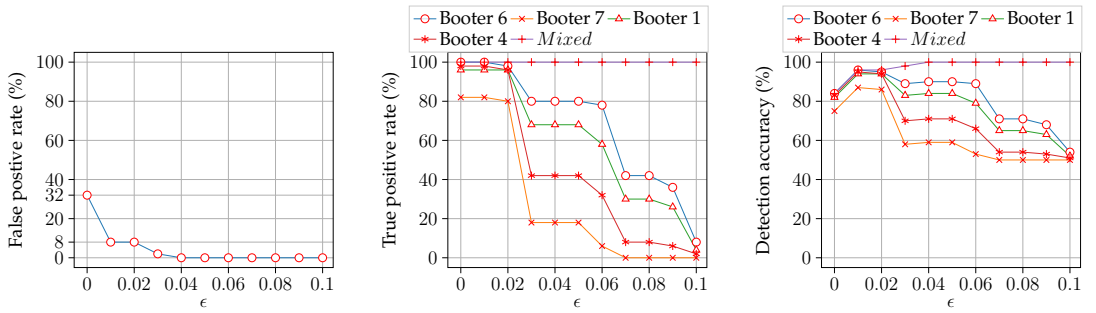
In this subsection, we evaluate our P4DDoS strategy against the state-of-the-art approach SOTA_DDoS in terms of D_{tp}, D_{fp} and D_{acc} in the case of Booter DDoS attacks. We also perform a sensitivity analysis of P4DDoS against the parameter ϵ, showing how the detection performance is affected by changing its value. The testing flow trace is composed by the concatenation of *Trace1* and *Trace2*: we first run 50-seconds legitimate flow trace (*Trace 1*) so that adaptive thresholds on entropy, for both strategies, are properly set in a legitimate traffic scenario. This trace allows us to evaluate D_{fp}. Then, *Trace 2* including different packet-rate DDoS attacks (also mixed), is used to evaluate D_{tp} and, together with results obtained in *Trace 1*, D_{acc}.

7.2.1 Comparison with the state of the art

To fairly compare P4DDoS with SOTA_DDoS, we tuned the sensitivity coefficient k of SOTA_DDoS (see [17]) to different

TABLE 3
Comparison of P4DDoS detection performance with a state-of-the-art approach [17] (Booter DDoS attacks)

Algorithm	False-positive rate D_{fp}	True-positive rate D_{tp} / Detection accuracy D_{acc}				
		Booter 6	Booter 7	Booter 1	Booter 4	Mixed
P4DDoS	8%	100% / 96%	82% / 87%	96% / 94%	98% / 95%	100% / 96%
SOTA_DDoS (k=5.5)	6%	96% / 95%	32% / 63%	62% / 78%	70% / 82%	100% / 97%
SOTA_DDoS (k=4.5)	8%	100% / 96%	38% / 65%	82% / 87%	78% / 85%	100% / 96%
SOTA_DDoS (k=3.5)	10%	100% / 95%	74% / 82%	100% / 95%	94% / 92%	100% / 95%
SOTA_DDoS (k=2.5)	20%	100% / 90%	94% / 87%	100% / 90%	100% / 90%	100% / 90%
SOTA_DDoS (k=1.5)	38%	100% / 81%	100% / 81%	100% / 81%	100% / 81%	100% / 81%
SOTA_DDoS (k=0.5)	60%	100% / 70%	100% / 70%	100% / 70%	100% / 70%	100% / 70%



(a) Sensitivity to false-positive rate (b) Sensitivity to true-positive rate (c) Sensitivity to detection accuracy

Fig. 6. Sensitivity analysis of P4DDoS to parameter ϵ

TABLE 4
Comparison of P4DDoS detection performance with a state-of-the-art approach [17] for different Botnet DDoS attack traffic proportions (ATPs)

Algorithm	False positive rate D_{fp}	True-positive rate D_{tp} / Detection accuracy D_{acc}					
		ATP: 5%	ATP: 10%	ATP: 15%	ATP: 20%	ATP: 25%	ATP: 30%
P4DDoS	8%	36% / 64%	92% / 92%	100% / 96%	100% / 96%	100% / 96%	100% / 96%
SOTA_DDoS (k=5.5)	6%	0% / 47%	12% / 53%	68% / 81%	100% / 97%	100% / 97%	100% / 97%
SOTA_DDoS (k=4.5)	8%	0% / 46%	40% / 66%	96% / 94%	100% / 96%	100% / 96%	100% / 96%
SOTA_DDoS (k=3.5)	10%	10% / 50%	50% / 70%	100% / 95%	100% / 95%	100% / 95%	100% / 95%
SOTA_DDoS (k=2.5)	20%	20% / 50%	88% / 84%	100% / 90%	100% / 90%	100% / 90%	100% / 90%
SOTA_DDoS (k=1.5)	38%	82% / 72%	94% / 78%	100% / 81%	100% / 81%	100% / 81%	100% / 81%
SOTA_DDoS (k=0.5)	60%	96% / 68%	100% / 70%	100% / 70%	100% / 70%	100% / 70%	100% / 70%

values: lower k leads to higher true-positive rate but also higher false-positive rate. Evaluation results are reported in Tab. 3. In the first 50 time intervals, four false alarms are detected by P4DDoS, being thus the false-positive rate 8%. As said, the false-positive rate of SOTA_DDoS increases as k decreases. False-positive rate of P4DDoS is slightly higher than of SOTA_DDoS only when $k = 5.5$ but, in that case, P4DDoS outperforms SOTA_DDoS on both true-positive rate and detection accuracy for all the considered Booter attacks. The best trade-off between all the metrics for SOTA_DDoS is obtained with $k = 3.5$. In this case, P4DDoS and SOTA_DDoS have comparable performance (with slightly better performance for P4DDoS). This means that, in this scenario, comparing the normalized entropy of destination IPs against a well-defined threshold is enough to get good performance on DDoS detection and that an evaluation of entropy of source IPs can be avoided (that is, same performance can be obtained with a simpler strategy).

7.2.2 Sensitivity analysis

Figure 6 reports the sensitivity of P4DDoS to normalized network traffic entropy parameter ϵ. Figure 6(a) shows that false-positive rate decreases as ϵ is smaller and stabilizes to zero once ϵ is larger than 0.04. This is because larger ϵ results in a smaller threshold, being more DDoS alarms triggered also when DDoS attacks are not occurring. False positives only happen for the legitimate traffic, reason why only one curve is reported. Figure 6(b) reveals the behavior of true-positive rate when ϵ varies, showing that in general true-positive rate decreases as ϵ increases. Figure 6(c) shows the impact of ϵ on detection accuracy. The shown curves, apart from the Mixed case, have a maximum at around $\epsilon = 0.01$: we then decided to set ϵ to this value, since it leads to the best trade-off considering all the three metrics.

7.3 Detection performance (Botnet DDoS attacks)

Table 4 shows a comparison on DDoS detection performance in case of internal Botnet DDoS attacks. The same methodology as described in Section 7.2 is adopted to prepare the testing flow trace but, in this case, *Trace1* and *Trace3* are concatenated. In this attack scenario, the cardinality of source IPs in the network does not change and the attack traffic proportion in Trace 3 is varied from 5% to 30%. Intuitively, the detection accuracy of P4DDoS increases

as the attack traffic proportion increases. When the attack traffic rate is low, i.e., 5%, the true-positive rate of P4DDoS is 36%. This is the drawback of most normalized entropy-based DDoS detection strategies: they struggle to detect low-packet-rate DDoS attacks since the normalized entropy may not significantly decrease. Nevertheless, our P4DDoS still has higher (or at least comparable) detection accuracy than SOTA_DDoS for any coefficient k. This is due to the fact that the entropy of destination IPs (not normalized) may decrease because of either a decrease in the cardinality of destination IPs in consecutive time intervals (see Section 2.1) or because a DDoS attack is occurring. Instead, the normalized entropy (used by P4DDoS) decreases only when a DDoS attack is occurring, since it is normalized to the cardinality of destination IPs. Thus, by considering non-normalized entropy as the metric to detect DDoS attacks as done by SOTA_DDoS, there is a higher chance of false positives due to legitimate traffic oscillations in consecutive time intervals. It is also important to note that the entropy of source IPs may not significantly increase when a Botnet DDoS attack occurs (as proven in [41]), so a simpler entropy-based DDoS detection system only considering normalized entropy of destination IPs may suffice for the detection of a wide range of attacks.

8 Related work

Here we recall existing works on *flow cardinality estimation* and on *entropy-based DDoS detection* in Software-Defined Networks with programmable data planes.

Flow cardinality estimation for network monitoring: Many cardinality-estimation algorithms have been implemented to be executed in programmable data planes for the purpose of network monitoring [5][6][16], often based on linear counting [39]. However, all of them are able to only perform the update operation directly in the data plane, while the query operation has still to be executed by the controller. This is because programmable switches do not support arithmetic operations such as logarithm and exponential function computation, which are needed for flow cardinality estimation. Conversely, by leveraging our proposed strategies for logarithm and exponential-function estimation in the data plane, named P4Log and P4Exp [1], we developed P4LogLog, a flow cardinality estimation algorithm that takes inspiration from LogLog [13]. P4LogLog enables a flow cardinality estimation entirely in programmable switches, where both update and query operations can be executed in the data plane. Moreover, our P4LogLog can estimate cardinality with high accuracy while consuming less memory than existing approaches. Note that HyperLogLog [24] has higher theoretical accuracy than LogLog, but it is currently not implementable in P4 language due to the computation of harmonic mean. HyperLogLog can also be implemented in CPU-based and FPGA-based programmable data planes [42]. However, the achievable throughput is limited and the adopted language is target-specific, while P4 can be used to program the data plane pipeline of heterogeneous hardware/software targets.

Entropy-based DDoS detection: Entropy-based DDoS detection has been widely studied in the context of SDN: a significant decrease in the (normalized) network entropy of destination IPs in a given time interval can be an indication of occurrence of a DDoS attack [2][3][4][43]. However, in most of previous works, entropy estimation is executed by the controller due to the complex way it is computed. Some works can be found in literature dealing with network traffic entropy estimation performed partially in the switches' data plane. For example, papers [6][7][16] all envision some operations to be executed by the programmable data plane, so that only summarized data must be sent to the controller. However, since the controller needs to frequently retrieve information from all the switches, the generated communication overhead is significant. Recently, Lapolli *et al.* [17] have demonstrated the feasibility of performing network traffic entropy estimation in the data plane using the P4 language, with the aim of detecting DDoS attacks. Their approach is valuable but it requires the usage of TCAM, which is instead avoided by our proposed P4DDoS. Moreover, P4DDoS and P4NEntropy adopt a time-based observation window, while [17] requires an observation window that includes a fixed power-of-two number of packets, making their solution less flexible. In fact, our approach may allow a controller to synchronize the retrieval of the estimated entropy from many programmable switches, paving the way towards the estimation of network traffic entropy on a *network-wide* scale [11] to improve the statistical relevance of monitored values.

9 Conclusion and future work

In this paper, relying on recently-proposed logarithmic and exponential function estimation solutions, we presented P4LogLog to estimate the number of distinct flows in the network by only using P4-supported operations. We then proposed P4NEntropy, a strategy that leverages P4LogLog for the estimation of normalized network traffic entropy directly in the switch's data plane. Finally, P4DDoS has been designed on top of P4NEntropy, with the goal of detecting DDoS attacks by means of an entropy-based system.

We also evaluated all of our proposed approaches and compared them with state-of-the-art solutions. Results show that P4LogLog has better accuracy than the state of the art especially when memory availability is small (i.e., smaller than 640 Bytes). Furthermore, P4NEntropy shows comparable accuracy on entropy estimation to existing approaches, but it leverages time-based observation windows (instead of fixed packet-based) and avoids the usage of TCAM (relying only on P4-supported operations). Finally, P4DDoS outperforms existing DDoS detection solutions implemented in P4 in terms of detection accuracy, especially in the case of internal Botnet DDoS attacks, while implementing a simpler logic. Moreover, unlike existing approaches in literature, all of our strategies avoid any communication overhead between controller and programmable switches, since they work entirely in the data plane. Specifically, P4DDoS only reports an alarm to the controller when an attack is detected.

As future work, we plan to extend our solution to detect other types of DDoS attacks, e.g. low-packet-rate, link flooding or carpet bombing attacks, with high accuracy. Furthermore, we also intend to work on an algorithm for the entropy-based detection of DDoS attacks on a network-wide scale, by collecting and combining the distributed entropy information from multiple programmable switches.

REFERENCES

[1] D. Ding, M. Savi, and D. Siracusa, "Estimating Logarithmic and Exponential Functions to Track Network Traffic Entropy in P4," in *IEEE/IFIP NOMS*, 2020.

[2] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments," *Computer Networks*, vol. 62, pp. 122–136, 2014.

[3] K. Kalkan, L. Altay, G. Gür, and F. Alagöz, "JESS: Joint Entropy-Based DDoS Defense Scheme in SDN," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2358–2372, 2018.

[4] R. Wang, Z. Jia, and L. Ju, "An entropy-based distributed DDoS detection mechanism in software-defined networking," in *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[5] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *USENIX NSDI*, 2013.

[6] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *ACM SIGCOMM*, 2016.

[7] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *ACM SIGCOMM*, 2017.

[8] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in Openflow software-defined networks," in *IEEE/IFIP NOMS*, 2014.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM Computer Communication Review*, vol. 38, no. 2, p. 69–74, 2008.

[10] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *ACM SOSR (SOSR)*, 2017.

[11] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "An Incrementally-Deployable P4-Enabled Architecture for Network-Wide Heavy-Hitter Detection," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 75–88, 2020.

[12] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[13] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *European Symposium on Algorithms*, 2003.

[14] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[15] "Behavioral model," https://github.com/p4lang/behavioral-model.

[16] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *ACM SIGCOMM*, 2018.

[17] A. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading Real-time DDoS Attack Detection to Programmable Data Planes," in *IFIP/IEEE IM*, 2019.

[18] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 1, pp. 145–156, 2006.

[19] H. S. Warren, "Hacker's delight," in *Pearson Education*, 2013.

[20] "The P4 Language Specification," https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[21] G. Cormode, "Count-min sketch," in *Encyclopedia of Database Systems*, pp. 511-516, 2009.

[22] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Springer ICALP*, 2002.

[23] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases*, 2011.

[24] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," *Discrete Mathematics and Theoretical Computer Science*, pp. 137–156, 2007.

[25] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *USENIX NSDI*, 2017.

[26] C. Wang, T. T. Miu, X. Luo, and J. Wang, "SkyShield: a sketch-based defense system against application layer DDoS attacks," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 559–573, 2018.

[27] "Mininet," http://mininet.org/.

[28] "Wireshark," https://www.wireshark.org/.

[29] D. Ding, M. Savi, F. Pederzolli, and D. Siracusa, "INVEST: Flow-Based Traffic Volume Estimation in Data-Plane Programmable Networks," in *IFIP Networking*, 2021.

[30] "P4LogLog," https://github.com/DINGDAMU/P4LogLog.

[31] "P4NEntropy," https://github.com/DINGDAMU/P4NEntropy.

[32] D. J. Struik, "The origin of L'Hopital's rule," *The Mathematics Teacher*, vol. 56, no. 4, pp. 257–260, 1963.

[33] "P4DDoS," https://github.com/DINGDAMU/P4DDoS.

[34] G. Nychis, V. Sekar, D. G. Andersen, H. Kim, and H. Zhang, "An Empirical Evaluation of Entropy-Based Traffic Anomaly Detection," in *ACM SIGCOMM Conference on Internet Measurement*, 2008.

[35] Y. Afek, A. Bremler-Barr, S. L. Feibish, and L. Schiff, "Detecting heavy flows in the SDN match and action model," *Computer Networks*, vol. 136, pp. 1–12, 2018.

[36] R. B. Basat, G. Einziger, S. L. Feibish, J. Moraney, and D. Raz, "Network-wide routing-oblivious heavy hitters," in *IEEE/ACM ANCS*, 2018.

[37] "Intel Tofino," https://www.barefootnetworks.com/products/brief-tofino/.

[38] "CAIDA UCSD anonymized Internet traces dataset," http://www.caida.org/data/passive/passive_dataset.xml.

[39] K. Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 208–229, 1990.

[40] J. Santanna, R. van Rijswijk-Deij, R. Hofstede, A. Sperotto, M. Wierbosch, L. Zambenedetti Granville, and A. Pras, "Booters - An analysis of DDoS-as-a-service attacks," in *IFIP/IEEE IM*, 2015.

[41] A. Bhandari, A. Sangal, and K. Kumar, "Destination address entropy based detection and traceback approach against distributed denial of service attacks," *International Journal of Computer Network and Information Security*, vol. 7, no. 8, p. 9, 2015.

[42] A. Kulkarni, M. Chiosa, T. B. Preußer, K. Kara, D. Sidler, and G. Alonso, "Hyperloglog sketch acceleration on FPGA," in *IEEE FPL*, 2020.

[43] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *IEEE ICNC*, 2015.

Damu Ding is a postdoctoral researcher at the University of Oxford. He received the Ph.D. degree in Electronics, Telecommunications and Information Technologies Engineering from the University of Bologna, Italy, co-tutored with Fondazione Bruno Kessler, Italy. His major research interests include SDN, network monitoring in P4-enabled programmable data planes, and vulnerability identification in programmable network devices.

Marco Savi is an assistant professor at University of Milano-Bicocca, Italy. He received his PhD degree in Information Technology (Telecommunication engineering) in 2016 from Politecnico di Milano and later he worked for four years at Fondazione Bruno Kessler, Trento, Italy. His research interests mainly focus on the design and optimization of telecommunication networks and on cloud computing. Dr. Savi has been involved in some European research projects advancing access and core network technologies.

Domenico Siracusa is the head of the RiSING research unit at Fondazione Bruno Kessler. He received the M.Sc. in Telecommunication Engineering (2008) and the Ph.D. in Information Technology (2012) from Politecnico di Milano. His research interests include SDN/NFV, cloud and fog computing, security and robustness. Domenico authored more than 90 publications appeared in international peer reviewed journals and in major conferences on networking technologies. Domenico acted as coordinator for both competitive EU-funded projects and commercial activities with important telecommunication vendors.